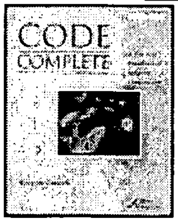
4



Chapter 19 - Self-Documenting Code

Code Complete: A Practical Handbook of Software Construction
by Steve McConnell
Microsoft Press © 1993

BACK

19.5 Commenting Techniques

Commenting is amenable to several different techniques depending on the level to which the comments app program, file, routine, paragraph, or individual line.

Commenting Individual Lines

In good code, the need to comment individual lines of code is rare. Here are two possible reasons a line of code would need a comment:

- The single line is complicated enough to need an explanation.
- The single line once had an error and you want a record of the error.

Here are some guidelines for commenting a line of code:

Avoid self-indulgent comments. Many years ago, I heard the story of a maintenance programmer who wa: called out of bed to fix a malfunctioning program. The program's author had left the company and couldn't be reached. The maintenance programmer hadn't worked on the program before, and after examining the documentation carefully, he found only one comment. It looked like this:

```
MOV AX. 723h     ; R. I. P. L. V. B.
```

After working with the program through the night and puzzling over the comment, the programmer made a successful patch and went home to bed. Months later, he met the program's author at a conference and four out that the comment stood for "Rest in peace, Ludwig van Beethoven." Beethoven died in 1827 (decimal), which is 723 (hexadecimal). The fact that 723h was needed in that spot had nothing to do with the comment. Aaarrrrghhhhh!

Endline comments and their problems

Endline comments are comments that appear at the ends of lines of code. Here's an example:

Basic Example of Endline Comments

```
for EmpID = 1 TO MaxRecords
   GetBonus( EmpID, EmpType, BonusAmt )
   if EmpType = Manager then
      PayMgrBonus( EmpID, BonusAmt )           ' pay intended, full amount
   elseif EmpType = Programmer then
      if BonusAmt >= MgrApprovalRequired then
         PayProgrBonus( EmpID, StdAmt() )      ' pay company std. amount
      else
         PayProgrBonus( EmpID, BonusAmt )      ' pay intended, full amount
      end if
next EmpID
```

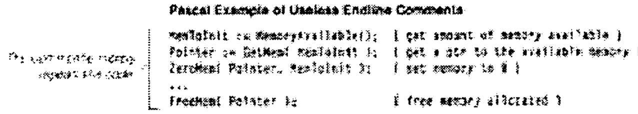
Although useful in some circumstances, endline comments pose several problems. The comments have to b aligned to the right of the code so that they don't interfere with the visual structure of the code. If you don't align them neatly, they'll make your listing look like it's been through the washing machine.

Endline comments tend to be hard to format. If you use many of them, it takes time to align them. Such time not spent learning more about the code; it's dedicated solely to the tedious task of pressing the spacebar or the tab key.

Endline comments are also hard to maintain. If the code on any line containing an endline comment grows, i bumps the comment farther out, and all the other endline comments will have to be bumped out to match. Styles that are hard to maintain aren't maintained, and the commenting deteriorates under modification rathe than improving.

Endline comments also tend to be cryptic. The right side of the line usually doesn't offer much room, and the desire to keep the comment on one line means that the comment must be short. Work then goes into makin the line as short as possible instead of as clear as possible. The comment usually ends up as cryptic as possible. You can eliminate this problem, unlike the others, with the use of 132-column monitors and printer

CODING HORROR **Avoid endline comments on single lines.** In addition to their practical problems, endline comments pose several conceptual problems. Here's an example of a set of endline comments:



Pascal Example of Useless Endline Comments

```
MemToInit := MemoryAvailable();   { get amount of memory available }
Pointer := GetMem( MemToInit );   { get a ptr to the available memory }
ZeroMem( Pointer, MemToInit );    { set memory to 0 }
...
FreeMem( Pointer );               { free memory allocated }
```

A systemic problem with endline comments is that it's hard to write a meaningful comment for one line of code. Most endline comments just repeat the line of code, which hurts more than it helps.

Avoid endline comments for multiple lines of code. If an endline comment is intended to apply to more than one line of code, the formatting doesn't show which lines the comment applies to. Here's an example:

Pascal Example of a Confusing Endline Comment on Multiple Lines of Code

```
for RateIdx := 1 to RateCount do begin    { Compute discounted rates }
   begin
   LookupRegularRate( RateIdx, RegularRate );
   Rate[ RateIdx ] := RegularRate * Discount[ RateIdx ];
   end;
```

Even though the content of this particular comment is fine, its placement isn't. You have to read the commen and the code to know whether the comment applies to a specific statement or to the entire loop.

When to use endline comments

Here are three exceptions to the recommendation against using endline comments:

CROSS-REFERENCE Other aspects of endline comments on data declarations are described in "Commenting Data Declarations," later in this section.

Use endline comments to annotate data declarations. Endline comments are useful for annotating data declarations because they don't have the same systemic problems as endline comments on code, provided that you have enough width. With 132 columns, you can usually write a meaningful comment beside each data declaration. Here's an example:

Pascal Example of Good Endline Comments for Data Declarations

```
Boundary:  Integer;    { upper index of sorted part of array }
InsertVal: String;     { data elmt to insert in sorted part of array }
InsertPos: Integer;    { position to insert elmt in sorted part of array }
```

Use endline comments for maintenance notes. Endline comments are useful for recording modifications t code after its initial development. This kind of comment typically consists of a date and the programmer's initials, or possibly an error-report number. Here's an example:

```
for i := 1 to MaxElmts-1 { fixed error #A423 10/1/92 (scm) }
```

Such a comment is handled better by version-control software, but if you don't have the tool support for version control and you want to annotate a single-line fix, this is the way to do it.

CROSS-REFERENCE Use of endline comments to mark ends of blocks is described further in "Commentinc Control Structures," later in this section.

Use endline comments to mark ends of blocks. An endline comment is useful for marking the end of a loi block of code—the end of a *while* loop or an *if* statement, for example. This is described in more detail later i this chapter.

Aside from a couple of special cases, endline comments have conceptual problems and tend to be used for code that's too complicated. They are also difficult to format and maintain. Overall, they're best avoided.

Commenting Paragraphs of Code

Most comments in a well-documented program are one- or two-sentence comments that describe paragraph of code. Here's an example:

C Example of a Good Comment for a Paragraph of Code

```
/* swap the roots */

OldRoot  = root[0];
root[0]  = root[1];
root[1]  = OldRoot;
```

The comment doesn't repeat the code. It describes the code's intent. Such comments are relatively easy to maintain. Even if you find an error in the way the roots are swapped, for example, the comment won't need t be changed. Comments that aren't written at the level of intent are harder to maintain.

Write comments at the level of the code's intent. Describe the purpose of the block of code that follows tl comment. Here's an example of a comment that's ineffective because it doesn't operate at the level of intent:

Pascal Example of an Ineffective Comment

```
{ check each character in "InputStr" until a dollar sign
  is found or all characters have been checked }

Done   := False;
MaxLen := Length( InputStr );
i      := 1;
while ( (not Done) and (i <= MaxLen) )
   begin
   if ( InputStr[ i ] = '$' ) then
      Done := True
   else
      i := i + 1
   end;
```

You can figure out that the loop looks for a *$* by reading the code, and it's somewhat helpful to have that summarized in the comment. The problem with this comment is that it merely repeats the code and doesn't give you any insight into what the code is supposed to be doing. This comment would be a little better:

```
{ find '$' in InputStr }
```

This comment is better because it indicates that the goal of the loop is to find a *$*. But it still doesn't give you much insight into why the loop would need to find a *$*—in other words, into the deeper intent of the loop. Here's a comment that's better still:

```
{ find the command-word terminator }
```

This comment actually contains information that the code listing does not, namely that the *$* terminates a command word. In no way could you deduce that merely from reading the code fragment, so the comment is genuinely helpful.

Another way of thinking about commenting at the level of intent is to think about what you would name a routine that did the same thing as the code you want to comment. If you're writing paragraphs of code that have one purpose each, it isn't difficult. The comment in the code above is a good example. *FindCommandWordTerminator()* would be a decent routine name. The other options, *Find$InInputString()* and *CheckEachCharacterInInputStrUntilADollarSignIsFoundOrAllCharactersHaveBeenChecked()*, are poor names for obvious reasons. Type the description without shortening or abbreviating it, as you might for a routine name. That description is your comment, and it's probably at the level of intent.

If the code is part of another routine, take the next step and put the code into its own routine. If it performs a well-defined function and you name the routine well, you'll add to the readability and maintainability of your code.

KEY POINT For the record, the code itself is always the first documentation you should check. In the case above, the literal, *$*, should be replaced with a named constant, and the variables should provide more of a clue about what's going on. If you want to push the edge of the readability envelope, add a variable to contain the result of the search. Doing that clearly distinguishes between the loop index and the result of the loop. Here's the code rewritten with good comments and good style:

Pascal Example of a Good Comment and Good Code

```
{ find the command-word terminator }

FoundTheEnd       := False;
MaxCommandLength  := Length( InputStr );
Idx               := 1;
while ( (not FoundTheEnd) and (Idx <= MaxCommandLength) )
   begin
   if ( InputStr[ Idx ] = COMMAND_WORD_TERMINATOR ) then
      begin
      FoundTheEnd  := True;
      EndOfCommand := Idx
      end
   else
      Idx := Idx + 1
   end;
```

If the code is good enough, it begins to read at close to the level of intent, encroaching on the comment's explanation of the code's intent. At that point, the comment and the code might become somewhat redundant, but that's a problem few programs have.

Focus paragraph comments on the *why* rather than the *how*. Comments that explain how something is done usually operate at the programming-language level rather than the problem level. It's nearly impossible for a comment that focuses on how an operation is done to explain the intent of the operation, and comments that tell how are often redundant. What does the following comment tell you that the code doesn't?

C Example of a Comment That Focuses on How

```
/* if allocation flag is zero */

if ( AllocFlag == 0 ) ...
```

CODING HORROR The comment tells you nothing more than the code itself does. What about this comment?

C Example of a Comment That Focuses on Why

```
/* if allocating new member */

if ( AllocFlag == 0 ) ...
```

This comment is a lot better because it tells you something you couldn't infer from the code itself. The code itself could still be improved by use of a meaningful named constant instead of *0*. Here's the best version of this comment and code:

C Example of Using Good Style in Addition to a "Why" Comment

```
/* if allocating new member */

if ( AllocFlag == NEW_MEMBER ) ...
```

Use comments to prepare the reader for what is to follow. Good comments tell the person reading the code what to expect. A reader should be able to scan only the comments and get a good idea of what the code does and where to look for a specific activity. A corollary to this rule is that a comment should always precede the code it describes. This idea isn't always taught in programming classes, but it's a well-establishe convention in commercial practice.

Make every comment count. There's no virtue in excessive commenting. Too many comments obscure the code they're meant to clarify. Rather than writing more comments, put the extra effort into making the code itself more readable.

Document surprises. If you find anything that isn't obvious from the code itself, put it into a comment. If you have used a tricky technique instead of a straightforward one to improve performance, use comments to poi out what the straightforward technique would be and quantify the performance gain achieved by using the tricky technique. Here's an example:

C Example of Documenting a Surprise

```
for ( i = 0; i < ElmtCount; i++ )
   {
   /* Use right shift to divide by two. Substituting the
      right-shift operation cuts the loop time by 75%. */

   Elmt[ i ] = Elmt[ i ] >> 1;
   }
```

The selection of the right shift in this example is intentional. Among experienced programmers, it's common knowledge that for integers, right shift is functionally equivalent to divide-by-two.

If it's common knowledge, why document it? Because the purpose of the operation is not to perform a right shift; it is to perform a divide-by-two. The fact that the code doesn't use the technique most suited to its purpose is significant. Moreover, most compilers optimize integer division-by-two to be a right shift anyway, meaning that the reduced clarity is usually unnecessary. In this particular case, the compiler evidently doesn optimize the divide-by-two, and the time saved will be significant. With the documentation, a programmer reading the code would see the motivation for using the nonobvious technique. Without the comment, the same programmer would be inclined to grumble that the code is unnecessarily "clever" without any meaning gain in performance. Usually such grumbling is justified, so it's important to document the exceptions.

Avoid abbreviations. Comments should be unambiguous, readable without the work of figuring out

abbreviations. Avoid all but the most common abbreviations in comments.

Unless you're using endline comments, using abbreviations isn't usually a temptation. If you are, and it is, realize that abbreviations are another strike against a technique that struck out several pitches ago.

Differentiate between major and minor comments. In a few cases, you might want to differentiate betwee different levels of comments, indicating that a detailed comment is part of a previous, broader comment. You can handle this in a couple of ways.

You can try underlining the major comment and not underlining the minor comment, as in the following:

C Example of Differentiating Between Major and Minor Comments with Underlines—Not Recommended

```
/* copy the string portion of the table, along the way
   omitting strings that are to be deleted */
/*--------------------------------------------------------------------------*/

/* determine number of strings in the table */

...

/* mark the strings to be deleted */

...
```

The weakness of this approach is that it forces you to underline more comments than you'd really like to. If you underline a comment, it's assumed that all the nonunderlined comments that follow it are subordinate to Consequently, when you write the first comment that isn't subordinate to the underlined comment, it too mus be underlined and the cycle starts all over. The result is too much underlining, or inconsistently underlining ir some places and not underlining in others.

This theme has several variations that all have the same problem. If you put the major comment in all caps and the minor comments in lowercase, you substitute the problem of too many all-caps comments for the problem of too many underlined comments. Some programmers use an initial cap on major statements and ı initial cap on minor ones, but that's a subtle visual cue that's too easily overlooked.

A better approach is to use ellipses in front of the minor comments. Here's an example:

C Example of Differentiating Between Major and Minor Comments with Ellipses

```
/* copy the string portion of the table, along the way
   omitting strings that are to be deleted */

/* ... determine number of strings in the table */

...

/* ... mark the strings to be deleted */

...
```

Another approach that's often best is to put the major-comment operation into its own routine. Routines shoɩ be logically "flat," with all their activities on about the same logical level. If your code differentiates between major and minor activities within a routine, the routine isn't flat. Putting the complicated group of activities int its own routine makes for two logically flat routines instead of one logically lumpy one.

This discussion of major and minor comments doesn't apply to indented code within loops and conditionals. such cases, you'll often have a broad comment at the top of the loop and more detailed comments about the operations within the indented code. In those cases, the indentation provides the clue to the logical organization of the comments. This discussion applies only to sequential paragraphs of code in which severa paragraphs make up a complete operation and some paragraphs are subordinate to others.

Comment anything that gets around an error or an undocumented feature in a language or an environment. If it's an error, it's probably not documented. Even if it's documented somewhere, it doesn't hu to document it again in your code. If it's an undocumented feature, by definition it isn't documented elsewher and it should be documented in your code.

Suppose you find that the library function *WriteData(Data, NumItems, BlockSize)* works properly except when *BlockSize* equals *500*. It works fine for *499*, *501* and every other value you've ever tried, but you have found that the routine has a defect that appears only when *BlockSize* equals *500*. In code that uses *WriteDa ()*, document why you're making a special case when *BlockSize* is *500*. Here's an example of how it could look:

Pascal Example of Documenting the Workaround for an Error

```
BlockSize := OptimalBlockSize( NumItems, SizePerItem );

{ The following code is necessary to work around an error in
  WriteData() that appears only when the third parameter
  equals 500. '500' has been replaced with a named constant
  for clarity. }

if ( BlockSize = WRITEDATA_BROKEN_SIZE )
   BlockSize := WRITEDATA_WORKAROUND_SIZE;

WriteData( File, Data, BlockSize );
```

Justify violations of good programming style. If you've had to violate good programming style, explain why. That will prevent a well-intentioned programmer from changing the code to a better style, possibly breaking your code. The explanation will make it clear that you knew what you were doing and weren't just sloppy—give yourself credit where credit is due!

Don't comment tricky code. One of the most prevalent and hazardous bits of programming folklore is that comments should be used to document especially "tricky" or "sensitive" sections of code. The reasoning is that people should know they need to be careful when they're working in certain areas.

FURTHER READING For other perspectives on writing good comments, see *The Elements of Programming Style* (Kernighan and Plauger 1978).

This is a scary idea.

Commenting tricky code is exactly the wrong approach to take. Comments can't rescue difficult code. As Kernighan and Plauger emphasize, "Don't document bad code—rewrite it" (1978).

HARD DATA One study found that areas of source code with large numbers of comments also tended to have the most defects and to consume the most development effort (Lind and Vairavan 1989). The authors hypothesized that programmers tended to comment difficult code heavily.

KEY POINT When someone says, "This is really *tricky* code," I hear them say, "This is really *bad* code." If something seems tricky to you, it'll be incomprehensible to someone else. Even something that doesn't seen all that tricky to you can seem impossibly convoluted to another person who hasn't seen the trick before. If yc have to ask yourself, "Is this tricky?", it is. You can always find a rewrite that's not tricky, so rewrite the code. Make your code so good that you don't need comments, and then comment it to make it even better.

This advice applies mainly to code you're writing for the first time. If you're maintaining a program and don't have the latitude to rewrite bad code, commenting the tricky parts is a good practice.

Commenting Data Declarations

CROSS-REFERENCE For details on formatting data, see "Laying Out Data Declarations" in Section 18.5.Fc details on how to use data effectively, see Chapters 8 through 12.

Comments for variable declarations describe aspects of the variable that the variable name can't describe. It important to document data carefully; at least one company that has studied its own practices has concludec that annotations on data are even more important than annotations on the processes in which the data is us((SDC, in Glass 1982). Here are some guidelines for commenting data:

Comment the units of numeric data. If a number represents length, indicate whether the length is express(in inches, feet, meters, or kilometers. If it's time, indicate whether it's expressed in elapsed seconds since 1-' 1980, milliseconds since the start of the program, and so on. If it's coordinates, indicate whether they represent latitude, longitude, and altitude and whether they're in radians or degrees; whether they represent an *X*, *Y*, *Z* coordinate system with its origin at the earth's center; and so on. Don't assume that the units are obvious. To a new programmer, they won't be. To someone who's been working on another part of the system, they won't be. After the program has been substantially modified, they won't be.

Comment the range of allowable numeric values. If a variable has an expected range of values, documer the expected range. If the language supports restricting the range—as Pascal and Ada do—restrict the rang(If not, use a comment to document the expected range of values. For example, if a variable represents an amount of money in dollars, indicate that you expect it to be between $1 and $100. If a variable indicates a voltage, indicate that it should be between 105v and 125v.

Comment coded meanings. If your language supports enumerated types—as Pascal, C, and Ada do—use them to express coded meanings. If it doesn't, use comments to indicate what each value represents—and use a named constant rather than a literal for each of the values. If a variable represents kinds of electrical current, comment the fact that *1* represents alternating current, *2* represents direct current, and *3* represents *undefined*.

Here's an example of documenting variable declarations that illustrates the three preceding recommendatior

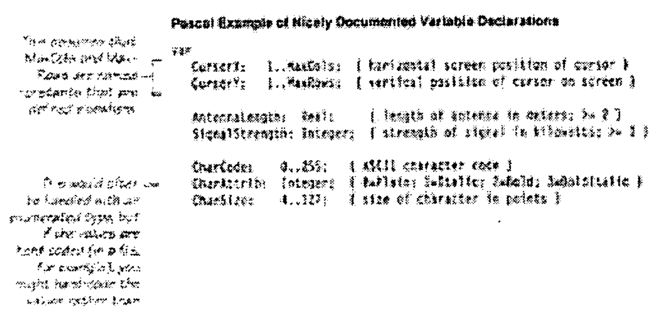
Basic Example of Nicely Documented Variable Declarations

```
DIM CursorX%   'horizontal cursor position; ranges from 1..MaxCols
DIM CursorY%   'vertical cursor position; ranges from 1..MaxRows

DIM AntennaLength!  'length of antenna in meters; range is >= 2
DIM SignalStrength% 'strength of signal in kilowatts; range is >= 1

DIM CharCode%     'ASCII character code; ranges from 0..255
DIM CharAttrib%   '0=Plain; 1=Italic; 2=Bold; 3=BoldItalic
DIM CharSize%     'size of character in points; ranges from 4..127
```

All the range information is given in comments. In a language that supports richer variable types, you can declare a few of the ranges with the variables. Here's an example:


Pascal Example of Nicely Documented Variable Declarations

```
var
   CursorX:   1..MaxCols;  { horizontal screen position of cursor }
   CursorY:   1..MaxRows;  { vertical position of cursor on screen }

   AntennaLength:  Real;     { length of antenna in meters; >= 2 }
   SignalStrength: Integer;  { strength of signal in kilowatts; >= 1 }

   CharCode:    0..255;   { ASCII character code }
   CharAttrib:  Integer;  { 0=Plain; 1=Italic; 2=Bold; 3=BoldItalic }
   CharSize:    4..127;   { size of character in points }
```

Comment limitations on input data. Input data might come from an input parameter, a file, or direct user input. The guidelines above apply as much to routine-input parameters as to other kinds of data. Make sure that expected and unexpected values are documented. Comments are one way of documenting that a routin

is never supposed to receive certain data. Assertions are another way, and if you can use them, the code becomes that much more self-checking.

Document flags to the bit level. If a variable is used as a bit field, document the meaning of each bit, as in the first example on the next page.

CROSS-REFERENCE For details on naming flag variables, see "Naming Status Variables" in Section 9.2.

Pascal Example of Documenting Flags to the Bit Level

```
var
    { The meanings of the bits in StatusFlags are as follows:

    MSB  0     error detected: 1=yes, 0=no
         1-2   kind of error: 0=syntax, 1=warning, 2=severe, 3=fatal
         3     reserved (should be 0)
         4     printer status: 1=ready, 0=not ready
         ...
         14    not used (should be 0)
    LSB  15    not used (should be 0) }

StatusFlags: integer;
```

If the example were written in C, it would call for bit-field syntax so that the bit-field meanings would be self-documenting.

Stamp comments related to a variable with the variable's name. If you have comments that refer to a specific variable, make sure that the comment is updated whenever the variable is updated. One way to improve the odds of a consistent modification is to stamp the comment with the name of the variable. That way, string searches for the variable name will find the comment as well as the variable.

CROSS-REFERENCE For details on using global data, see Section 10.6, "Global Variables."

Document global data. If global data is used, annotate each piece well at the point at which it is declared. The annotation should indicate the purpose of the data and why it needs to be global. At each point at which the data is used, make it clear that the data is global. A naming convention is the first choice for highlighting variable's global status. If a naming convention isn't used, comments can fill the gap.

Commenting Control Structures

CROSS-REFERENCE For other details on control structures, see Section 18.3, "Layout Styles," Section 18. "Laying Out Control Structures," and Chapters 13 through 17.

The space before a control structure is usually a natural place to put a comment. If it's an *if* or a *case* statement, you can provide the reason for the decision and a summary of the outcome. If it's a loop, you can indicate the purpose of the loop. Here are a couple of examples:

C Example of Commenting the Purpose of a Control Structure

Purpose of the following loop

```
/* copy input field up to comma */

while ( *InputStr != ',' && *InputStr != END_OF_STRING )
   {
```

```
      *field = *inputStr;
      field++;
      inputStr++;
      } /* while -- copy input field */

   *field = END_OF_STRING;

   /* if at end of string, all actions are complete */

   if ( *inputStr != END_OF_STRING )
      {
      /* read past comma and subsequent blanks to get to the
         next input field */

      *inputStr++;
      while ( *inputStr == ' ' && *inputStr != END_OF_STRING )
         inputStr++;
      } /* if -- at end of string */
```

This example suggests some guidelines.

Put a comment before each block of statements, *if*, *case*, or loop. Such a place is a natural spot for a comment, and these constructs often need explanation. Use a comment to clarify the purpose of the control structure.

Comment the end of each control structure. Use a comment to show what ended—for example,

```
end; { for ClientIdx -- process record for each client }
```

CROSS-REFERENCE For details on naming loops, see "Exiting the Loop" in Section 15.2.

A comment is especially helpful at the end of long or nested loops. In languages that support named loops (Ada, for example), name the loops. In other languages, use comments to clarify loop nesting. Here's a Pasc example of using comments to clarify the ends of loop structures:

Pascal Example of Using Comments to Show Nesting

```
for TableIdx := 1 to TableCount
   begin
   while ( RecordIdx < RecordCount )
      begin
      if ( not IllegalRecNum( RecordIdx ) )
         begin
         ...
         end; { if }
      end; { while }
   end; { for }
```

This commenting technique supplements the visual clues about the logical structure given by the code's indentation. You don't need to use the technique for short loops that aren't nested. When the nesting is deep or the loops are long, however, the technique pays off.

Even though it's worthwhile, putting the comments in and maintaining them can get tedious, and the best wa to avoid the tedious work is often to rewrite the complicated code that requires tedious documentation.

Commenting Routines

CROSS-REFERENCE For details on formatting routines, see Section 18.7, "Laying Out Routines." For detai on how to create high-quality routines, see Chapter 5, "Characteristics of High-Quality Routines."

Routine-level comments are the subject of some of the worst advice in typical computer-science textbooks. Many textbooks urge you to pile up a stack of information at the top of every routine, regardless of its size or complexity. Here's an example:

CODING HORROR

Basic Example of a Monolithic, Kitchen-Sink Routine Prolog

```
'**********************************************************************
```

```
'
' Name: CopyString
'
' Purpose:      This routine copies a string from the source
'               string (Source$) to the target string (Target$).
'
' Algorithm:    It gets the length of Source$ and then copies each
'               character, one at a time, into Target$. It uses
'               the loop index as an array index into both Source$
'               and Target$ and increments the loop/array index
'               after each character is copied.
'
' Inputs:       Input$    The string to be copied
'
' Outputs:      Output$   The string to receive the copy of Input$
'
' Interface Assumptions: None
'
' Modification History: None
'
' Author:       Dwight K. Coder
' Date Created: 10/1/92
' Phone:        (222) 555-2255
' SSN:          111-22-3333
' Eye Color:    Green
' Maiden Name:  None
' Blood Type:   AB-
' Mother's Maiden Name: None
'
'**********************************************************************
```

This is ridiculous. *CopyString* is presumably a trivial routine—probably fewer than five lines of code. The comment is totally out of proportion to the scale of the routine. The parts about the routine's *Purpose* and *Algorithm* are strained because it's hard to describe something as simple as *CopyString* at a level of detail that's between "copy a string" and the code itself. The boilerplate comments *Interface Assumptions* and *Modification History* aren't useful either—they just take up space in the listing. To require all these ingredient for every routine is a recipe for inaccurate comments and maintenance failure. It's a lot of make-work that never pays off. Here are some guidelines for commenting routines:

Keep comments close to the code they describe. One reason that the prolog to a routine shouldn't contai voluminous documentation is that such a practice puts the comments far away from the parts of the routine they describe. During maintenance, comments that are far from the code tend not to be maintained with the code. The comments and the code start to disagree, and suddenly the comments are worthless.

Instead, follow the Principle of Proximity and put comments as close as possible to the code they describe. They're more likely to be maintained, and they'll continue to be worthwhile.

Several components of routine prologs are described below and should be included as needed. For your convenience, create a boilerplate documentation prolog. Just don't feel obliged to include all the information every case. Fill out the parts that matter and delete the rest.

CROSS-REFERENCE Good routine names are key to routine documentation. For details on how to create them, see Section 5.2, "Good Routine Names."

Describe each routine in one or two sentences at the top of the routine. If you can't describe the routine in a short sentence or two, you probably need to think harder about what it's supposed to do. Difficulty in creating a short description is a sign that the design isn't as good as it should be. Go back to the design drawing board and try again. The short summary statement should be present in virtually all routines.

Document input and output variables where they are declared. If you're not using global data, the easies

way to document input and output variables is to put comments next to the parameter declarations. Here's ai example:

Pascal Example of Documenting Input and Output Data Where It's Declared—Good Practice

```
procedure InsertionSort
   (
   Var Data:      tSortArray; { sort array elements FirstElmt..LastElmt }
       FirstElmt: Integer;    { index of first element to sort }
       LastElmt: Integer      { index of last element to sort }
   );
```

CROSS-REFERENCE Endline comments are discussed in more detail in "Endline comments and their problems," earlier in this section.

This practice is a good exception to the rule of not using endline comments; they are exceptionally useful in documenting input and output variables. This occasion for commenting is also a good illustration of the valuε of using standard indentation rather than endline indentation for routine parameter lists; you wouldn't have room for meaningful endline comments if you used endline indentation. The comments in the example are strained for space even with standard indentation, although it would be less a problem if the code in this boo had more than 80 columns to work with. This example also demonstrates that comments aren't the only forrr of documentation. If your variable names are good enough, you might be able to skip commenting them. Finally, the need to document input and output variables is a good reason to avoid global data. Where do yo document it? Presumably, you document the globals in the monster prolog. That makes for more work and unfortunately in practice usually means that the global data doesn't get documented. That's too bad becausε global data needs to be documented at least as much as anything else.

Differentiate between input and output data. It's useful to know which data is used as input and which is used as output. Pascal makes it relatively easy to tell because output data is preceded by the *var* keyword ai input data isn't. If your language doesn't support such differentiation automatically, put it into comments. Here's an example in C:

CROSS-REFERENCE The order of these parameters follows the standard order for C routines but conflicts with more general practices. For details, see "Put parameters in inputmodify-output order" in Section 5.7.For details on using a naming convention to differentiate between input and output data, see Section 9.3, "The Power of Naming Conventions."

C Example of Differentiating Between Input and Output Data

```
void StringCopy
   (
   char *   Target,       /* out: string to copy to */
   char *   Source        /* in:  string to copy from */
   )
...
```

C-language routine declarations are a little tricky because some of the time the asterisk (*) indicates that the argument is an output argument, and a lot of the time it just means that the variable is easier to handle as a pointer than as a base type. You're usually better off identifying input and output arguments explicitly.

If your routines are short enough and you maintain a clear distinction between input and output data, documenting the data's input or output status is probably unnecessary. If the routine is longer, however, it's ε useful service to anyone who reads the routine.

CROSS-REFERENCE For details on other considerations for routine interfaces, see Section 5.7, "How to Us Routine Parameters."

Document interface assumptions. Documenting interface assumptions might be viewed as a subset of the other commenting recommendations. If you have made any assumptions about the state of variables you receive—legal and illegal values, arrays being in sorted order, data structures being initialized or containing only good data, and so on—document them either in the routine prolog or where the data is declared. This

documentation should be present in virtually every routine.

Make sure that global data that's used is documented. A global variable is as much an interface to a routine . anything else and is all the more hazardous because it sometimes doesn't seem like one.

As you're writing the routine and realize that you're making an interface assumption, write it down immediate

CROSS-REFERENCE For details on the fact that certain routines have a disproportionate number of errors, see "Which Routines Contain the Most Errors?" in Section 25.4.

Keep track of the routine's change history. Keep track of changes made to a routine after its initial construction. Changes are often made because of errors, and errors tend to be concentrated in a few troublesome routines. A lot of errors in a routine means that the same routine probably has even more errors Keeping track of the errors detected in a routine means that if the number of errors reaches a certain point, you'll know it's time to redesign and rewrite the routine.

This isn't something you would expect to do for every routine, mainly because you wouldn't expect to see errors in every routine. Tracking a routine's change history can result in an annoying clutter of error-fix descriptions at the top of a routine, but the approach is self-correcting. If you get too many errors clogging up the top of the routine, your natural inclination will be to get rid of the distraction and rewrite the routine. Most programmers enjoy having an excuse to rewrite code they know could have been written better anyway, and this documentation is a good, visible justification for doing so.

Comment on the routine's limitations. If the routine provides a numeric result, indicate the accuracy of the result. If the computations are undefined under some conditions, document the conditions. If the routine has default behavior when it gets into trouble, document the behavior. If the routine is expected to work only on arrays or tables of a certain size, indicate that. If you know of modifications to the program that would break the routine, document them. If you ran into gotchas during the development of the routine, document them tc

Document the routine's global effects. If the routine modifies global data, describe exactly what it does to the global data. As mentioned in Section 5.4, modifying global data is at least an order of magnitude more dangerous than merely reading it, so modifications should be performed carefully, part of the care being clea documentation. As usual, if documenting becomes too onerous, rewrite the code to reduce the use of global data.

Document the source of algorithms that are used. If you have used an algorithm from a book or magazin document the volume and page number you took it from. If you developed the algorithm yourself, indicate where the reader can find the notes you've made about it.

Use comments to mark parts of your program. Some programmers use comments to mark parts of their program so that they can find them easily. One such technique in C is to mark the top of each routine with a comment such as

```
/***
```

This allows you to jump from routine to routine by doing a string search for /***.

A similar technique is to mark different kinds of comments differently, depending on what they describe. For example, in Pascal you could use *{-X-*, where *X* is a code you use to indicate the kind of comment. The comment *{-R-* could indicate that the comment describes a routine, *{-I-* input and output data, *{-L-* local data descriptions, and so on. This technique allows you to use tools to extract different kinds of information from your source files. For example, you could search for *{-R-* to retrieve descriptions of all the routines.

Commenting Files, Modules, and Programs

CROSS-REFERENCE For layout details, see Section 18.8, "Laying Out Files, Modules, and Programs." For details on using modules, see Chapter 6, "Three out of Four Programmers Surveyed Prefer Modules."

Files, modules, and programs are all characterized by the fact that they contain multiple routines. A file or module should contain a collection of related routines. A program contains all the routines in a program. The documentation task in each case is to provide a meaningful, top-level view of the contents of the file, module or program. The issues are similar in each case, so I'll just refer to documenting "files," and you can assume that the guidelines apply to modules and programs as well.

General guidelines for file documentation

At the top of a file, use a block comment to describe the contents of the file. Here are some guidelines for the block comment:

Describe the purpose of each file. If all the routines for a program are in one file, the purpose of the file is pretty obvious. If you're working on a program with multiple modules and the modules are separated into multiple files, explain why certain routines are put into certain files. Describe the purpose of each file. Since one file equals one module in this case, that will be sufficient documentation.

If the division into multiple source files is made for some reason other than modularity, a good description of the purpose of the file will be even more helpful to a programmer who is modifying the program. If someone i looking through the program's files for a routine that does *x*, can he or she tell whether this file contains such routine?

Put your name and phone number in the block comment. Authorship is important information to have in a listing. It gives other programmers who work on the code a clue about the programming style, and it gives them someone to call if they need help. Depending on whether you work on individual routines, modules, or programs, you should include author information at the routine, module, or program level.

Include a copyright statement in the block comment. Many companies like to include copyright statemen in their programs. If yours is one of them, include a line similar to this one:

C Example of a Copyright Statement

```
/* (c) Copyright 1993 Steve McConnell, Inc. All Rights Reserved. */
...
```

The Book Paradigm for program documentation

Most experienced programmers agree that the documentation techniques described in the previous section are valuable. The hard, scientific evidence for the value of any one of the techniques is still weak. When the techniques are combined, however, evidence of their effectiveness is strong.

FURTHER READING This discussion is adapted from "The Book Paradigm for Improved Maintenance" (Oman and Cook 1990a) and "Typographic Style Is More Than Cosmetic" (Oman and Cook 1990b). A similar analysis is presented in detail in *Human: Factors and Typography for More Readable Programs* (Baecker and Marcus 1990).

In 1990, Paul Oman and Curtis Cook published a pair of studies on the "Book Paradigm" for documentation (1990a, 1990b). They looked for a coding style that would support several different styles of code reading. One goal was to support top-down, bottom-up, and focused searches. Another was to break up the code intc chunks that programmers could remember more easily than a long listing of homogeneous code. Oman and Cook wanted the style to provide for both high-level and low-level clues about code organization.

They found that by thinking of code as a special kind of book and formatting it accordingly, they could achiev their goals. In the Book Paradigm, code and its documentation are organized into several components simila to the components of a book to help programmers get a high-level view of the program.

The "preface" is a group of introductory comments such as those usually found at the beginning of a file. It functions as the preface to a book does. It gives the programmer an overview of the program.

The "table of contents" shows the files, modules, and routines (chapters). They might be shown in a list, as a traditional book's chapters are, or graphically, in a structure chart.

The "sections" are the divisions within routines—routine declarations, data declarations, and executable statements, for example.

The "cross-references" are cross-reference maps of the code, including line numbers.

The low-level techniques that Oman and Cook use to take advantage of the similarities between a book and code listing are similar to the techniques described in Chapter 18, "Layout and Style," and in this chapter.

HARD DATA The upshot of using their techniques to organize code was that when Oman and Cook gave a maintenance task to a group of experienced, professional programmers, the average time to perform a maintenance task in a 1000-line program was only about three-quarters of the time it took the programmers i do the same task in a traditional source listing (1990b). Moreover, the maintenance scores of programmers c code documented with the Book Paradigm averaged about 20 percent higher than on traditionally document code. Similar results were obtained using both Pascal and C. Oman and Cook concluded that by paying attention to the typographic principles of book design, you can get a 10 to 20 percent improvement in comprehension. A study with student programmers at the University of Toronto produced similar results (Baecker and Marcus 1990).

The Book Paradigm emphasizes the importance of providing documentation that explains both the high-leve and the low-level organization of your program.

CHECKLIST Good Commenting Technique

GENERAL

- Does the source listing contain most of the information about the program?
- Can someone pick up the code and immediately start to understand it?
- Do comments explain the code's intent or summarize what the code does, rather than just repeating the code?
- Is the PDL-to-code process used to reduce commenting time?
- Has tricky code been rewritten rather than commented?
- Are comments up to date?
- Are comments clear and correct?
- Does the commenting style allow comments to be easily modified?

STATEMENTS AND PARAGRAPHS

- Does the code avoid endline comments?
- Do comments focus on *why* rather than *how*?
- Do comments prepare the reader for the code to follow?
- Does every comment count? Have redundant, extraneous, and self-indulgent comments been removed or improved?
- Are surprises documented?
- Have abbreviations been avoided?
- Is the distinction between major and minor comments clear?
- Is code that works around an error or undocumented feature commented?

DATA DECLARATIONS

- Are units on data declarations commented?
- Are the ranges of values on numeric data commented?

- Are coded meanings commented?
- Are limitations on input data commented?
- Are flags documented to the bit level?
- Has each global variable been commented where it is declared?
- Has each global variable been identified as such at its use, by either a naming convention or a comment?
- Are magic numbers documented or, preferably, replaced with named constants or variables?

CONTROL STRUCTURES

- Is each control statement commented?
- Are the ends of long or complex control structures commented?

ROUTINES

- Is the purpose of each routine commented?
- Are other facts about each routine given in comments, when relevant, including input and output dat interface assumptions, limitations, error corrections, global effects, and sources of algorithms?

FILES, MODULES, AND PROGRAMS

- Does the program have a short document such as that described in the Book Paradigm that gives a overall view of how the program is organized?
- Is the purpose of each file described?
- Are the author's name and phone number in the listing?

BACK